Fachhochschule
Münster University of
Applied Sciences

Java Closures

Die Entwicklung von C *function-pointers*, C++ *function-templates*, Java *anonymos-functions*, hin zu C# *delegates* und Java *Closures*.

Prof. Dr. Nikolaus Wulff

Funktionszeiger

Zeiger auf Funktionen – anstatt auf Daten, Klassen oder Strukturen – sind ein wichtiges Programmiermittel.

- Sie erleichtern es generische Algorithmen zu entwickeln und
- Objekte mit Callback Methoden zu parameterisieren.
- ***C*** verwendet echte Funktionszeiger.
- ***C***$^{++}$ zusätzlich Function Templates.
- Sprachen wie ***C#*** oder *Groovy* verwenden Delegates und Closures.
- Java ab der Version 1.8 enthält Closures.

C Function-Pointer

- Funktionszeiger erlauben allgemeine Algorithmen. Sollen z.B. *sort* oder *max* für beliebige Datentypen entwickelt werden so kann diese universell geschehen, sobald klar ist wie die Ordnungsrelation < (cmp) und eine evt. Vertausche-Funktion (swap) passend zum ADT zu entwickeln sind.
- Der Prototyp einer generischen max Funktion in C könnte daher wie folgt definiert werden:

```
void* max(int len,          // # number of  elements
void* array,                // pointer to the elements
size_t size,                // sizeof one element
int (*cmp)(const void*, const void*));
```

C Syntax Zucker

- Mit einigen typedef Definitionen wird die Prototyp-Signatur besser lesbar:

```
typedef const void* Object;
typedef unsigned int uint;

typedef int (*Comperator) (Object x, Object y);

Object maximum(uint len, size_t size,
               Object array, Comperator cmp);
```

- So lassen sich generische Min/Max/Sort Algorithmen für beliebige Datentypen entwickeln...

C++ template Funktionen

- C++ erlaubt es sowohl Klassen, als auch Funktionen als Template zu entwickeln:

```
template<class T>
bool cmp(const T& a, const T& b) {
    return a<b;
}
```

```
template<class T>
T maximum(int len, T v[]) {
   int j,k=0;
   for(j=1;j<len;j++) {
      if(cmp(v[k],v[j])) {
         k = j;
      }
   }
   return v[k];
}
```

Ähnlich zu Java Generics werden hier typisierte Methoden an einen Typ **T** gebunden...

Falls **T** keinen < Operator definiert, so compiliert der Aufruf der maximum Funktion nicht...

Funktions Template vers. Generic

- Ein C++ Template gehört in eine C++ Header-Datei.
- Der C++ Compiler generiert statisch passend zum Typen **T** die implementierende Funktion innerhalb des aufrufenden Moduls.
- Ungeschickt eingesetzt kann dies zu einem größeren Binärcode führen, da unter Umständen die selbe Funktion in vielen verschiedenen Load-Modulen vorkommen kann und übersetzt wird!
- Im Gegensatz dazu werden Java Generics nur einmal übersetzt und per type-erasure ohne Typinformation als eine einzige Klasse abgespeichert.

cmp Funktion überladen

- Für ein Color-struct kann eine spezielle cmp-Funktion definiert werden:

```
typedef struct color_struct {
    unsigned char r,g,b;
} Color;
```

```
template<class Color>
bool cmp(Color& a, Color& b){
   int x = value(a);
   int y = value(b);
   return x < y;
}
```

Neudefinition von cmp für den Typ Color...

- Jetzt weiß der Compiler welche cmp Funktion zu wählen ist, oder aber Color muss als Klasse mit einem überladenen kleiner Operator < definiert sein.

C# Delegate

- C# bietet per ***delegate*** echte typisierte Referenzen ähnlich den C Funktionszeigern an.

```
delegate bool Comperator<T>(T a, T b);

class Maximizer {

   public static T maximize<T>(T[] v, Comperator<T> cmp) {
       int j,k=0;
       for(j=1;j<v.Length;j++) {
          if(cmp(v[k],v[j])) {
             k = j;
          }
       }
       return v[k];
   }
}
```

C# Delegate Verwendung

- Der Methodenzeiger wird vom Compiler automatisch ermittelt und als Delegate verwendet:

```
public static void Main(string[] args){
   double[] dfield = new double[]{3.0,23,2.0,-25,5.1};

   double dmax = Maximizer.maximize<double>(dfield,compare);
   Console.WriteLine("max: " + dmax);

}

public static bool compare(double a, double b) {
      return a < b;
}
```

Groovy Closure

- Groovy verwendet das Closure Konstrukt, dessen Definition an das mathematische λ-Kalkül erinnert:

```
static <T> T maximize(List<T> v, Closure cmp) {
     int j,k=0
     for(j=1; j< v.size(); j++) {
         if(cmp(v[k],v[j])) {
             k = j
         }
     }
     return v[k]
}

static main(args) {
    def dfield = [1.0,3.34, 435,-6]
    def comparer =  { a, b -> return a < b }

    double dmax = maximize(dfield, comparer)
    print("max: " + dmax)
}
```

Java Generic Implementierung

- Eine analoge maximum Funktion als Java Generic:

```
public interface Comperator<T> {
    boolean cmp(final T a, final  T b);
}

public class Maximizer<T> {
    public static <T> T max(final T[] v, final Comperator<T> fp) {
        int k=0, len = v.length;
        for(int j=1;j<len;j++) {
            if(fp.cmp(v[k],v[j])) {
               k = j;
            }
        }
        return v[k];
    }
```

- Hier wird die Vergleichsoperation per Schnittstelle übergeben (Ähnliches wäre auch in C++ möglich...)

Anonymous Class

- Der Vergleich könnte mit einer inneren anonymen Implementierung erfolgen:

```
Double[] dfield= {1.,2.,8.,-4.,-16.,15.};

double dmax = Maximizer.max(dfield, new Comperator<Double>() {
    @Override
    public boolean cmp(Double a, Double b) {
        return a.doubleValue() < b.doubleValue();
    }
});
```

- Anonyme Klassen waren bis JDK1.7 der „Java Weg“ für anders nicht zu realisierende C Funktionszeiger.
- Viele Callback-Funktionen in Swing basierten GUI Applikationen waren mit AK implementiert.

Java Closure

- Java Closures sehen eine Syntax ähnlich zu Groovy vor mit -> als Closure Operator. Dann kann anstatt einer anonymen Klasse stehen:

```
double dmax = Maximizer.max(dfield,(a,b) -> {return a < b;} );

double dmax = Maximizer.max(dfield,(a,b) -> a < b );
```

- Falls dies der einzige Unterschied zwischen Closures und anonymen Klassen wäre, würde sich der ganze Aufwand nicht lohnen. Weitere Nachteile sind:
  - Java's anonyme Klassen können nicht auf lokale Variablen der äußeren Methode zugreifen (außer bei final).
  - Der this-Zeiger der einhüllenden Klasse ist nicht im Scope.
  - Überladene Methoden sind nicht aufrufbar.

Groovy: Collection filtern

- Das Filtern einer Collection ist mit Closures sehr einfach und elegant zu implementieren:

```
static <T> List<T> filter(List<T> v, Closure cond) {
    List<T> ret = new ArrayList<T>();
    for(T t in v) {
        if(cond(t)) ret.add(t);
    }
    return ret;
}

static main(args) {
    def list = [1.0,3.34, 435,-6,8]

    def positv = filter(list, {y->return 0<y})  // Closure inline
    println("positv: " + positv);
    def x = 5

    def greater = { a -> return x<a }
    def large = filter(list, greater)           // gebunden an ref
    println("subset: " + large);
}
```

Java Filter

```
interface Condition<T> {
    boolean match(T args);
}

static <T> List<T> filter(List<T> v, Condition<T> cond) {
    List<T> ret = new ArrayList<T>();
    for(T t: v) {
        if(cond.match(t)) ret.add(t);
    }
    return ret;
}

public static void main(String[] args) {
    List<Double> list = Arrays.asList(-2.1,1., 3.3,43.0,-6.0,8.0);
    List<Double> positv = filter(list, new Condition<Double>() {
        @Override
        public boolean match(Double v) {     // anonymous Class
            return v.doubleValue()>0;        // quasi inline
        }
    });
    System.out.println("positv: " + positv);
```

Lokale Variable nicht im Scope

- Lokale Variablen sind in anonymen Klassen nicht sichtbar:

```
double x_min = 4;
List<Double> positv = filter(list, new Condition<Double>() {
    @Override
    public boolean match(Double v) {  // anonymous Class
        return v.doubleValue()>x_min; // error x_min nicht im scope
    }
});
```

Seit JDK V1.8
effectively final

- Damit auf x_min zugegriffen werden kann muss die lokale Variable als ***final deklariert*** werden!
- Oder es muss eine extra Klasse entwickelt werden...

Filter Klasse

- Eine eigene Filter Klasse lohnt sich nur, wenn deren Funktionalität häufiger erforderlich ist:

```
public class GreaterThan implements Condition<Double> {
    private double x_min;
    public GreaterThan(double x) {
        x_min = x;
    }
    public void setXmin(double x) {
        x_min = x;
    }

    @Override
    public boolean match(Double v) {
        return v.doubleValue()>x_min;
    }
}
```

- Insgesamt sind Closures wesentlich flexibler und universeller einsetzbarer...

Überladene Methoden

- Überladene Methoden der äußeren Klasse sind nicht im Scope der anonymen Klasse:

```
public String toString(String fmt, Object ...args) {
    return String.format(fmt, args);
}

public void test() {
    String s = "lokale Variable";

    Runnable printer = new Runnable() {
        @Override
        public void run() {
            String x = toString("%s \n",s);    // compile error
            System.out.println(x);
        }
    };
    printer.run();

}
```

Klimmzüge

- Der this-Zeiger muss als final übergeben werden, um die überladene toString Methode zu erreichen...

```
public String toString(String fmt, Object ...args) {
    return String.format(fmt, args);
}

public void test() {
    String s = "lokale Variable";
    final String final_s = s;
    final AnonymousExample self = this;

    Runnable printer = new Runnable() {
        @Override
        public void run() {
            String x = self.toString("%s \n",final_s);
            System.out.println(x);
        }
    };
    printer.run();
```

Java1.8: Closure Filter

```
@FunctionalInterface
interface Condition<T> {        // @see java.util.function.Predicate<T>
    boolean match(T args);
}

static <T> List<T> filter(List<T> v, Condition<T> cond) {
    List<T> ret = new ArrayList<T>();
    for(T t: v) {
        if(cond.match(t)) ret.add(t);
    }
    return ret;
}

public static void main(String[] args) {
    List<Double> list = Arrays.asList(-2.1,1., 3.3,43.0,-6.0,8.0);
    List<Double> range= filter(list, it -> 0 < it && it < 5);
    System.out.println("0..5: " + range);
```

Java: Closures mit Zustand

- Das erzeugte Closure Objekt kann auf die äußeren lokalen Variable zugreifen ohne das diese explizit final sind (sie sind *effectively final*):

Seit JDK1.8
effectively final

```
@Test
public void testFilter() {
    int min = 0, max = 3;
    List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5, 6);
    List<Integer> expected = Arrays.asList(1, 2, 3);

    Collection<Integer> ret = cf.filter(ints,
                                        y -> min <= y && y <= max);

    Assert.assertEquals(expected, ret);
}
```

- Nicht nur das Closure Objekt ist an die filter Methode übergeben, sondern auch der min und max Wert.

Closures die Zeit ist reif

- Anonyme Klassen sind ein schlechter Ersatz für Closures, da ihnen wesentliche Eigenschaften fehlen.
- Closures kapseln lauffähige Funktionen als echte Objekte, sogenannte Functors mit eigenem Status.
- Geschickt eingesetzt und implementiert können Functors asynchron ausgeführt werden und erlauben eine multi-threaded Ausführung.
- Seit JDK1.8 gehören Closure zu Java und vieles wird einfacher, insbesondere mit zahlreichen „default“ Implementierungen in interfaces und vordefinierten *@FunctionalInterfaces* im Paket *java.util.function*.